# 2024 秋《计算机硬件基础》

## 09 链接与运行 作业

CSAPP 7. 7 不改变任何变量名字，不改变原有赋值语句，修改 7. 6. 1 节中的 bar5.c, 使得 foo5.c 输出 x 和 y 的正确值（也就是整数 15213 和 15212 的十六进制表示）。

```
1   /* foo5.c */
2   #include <stdio.h>
3   void f(void);
4
5   int y = 15212;
6   int x = 15213;
7
8   int main()
9   {
10      f();
11      printf("x = 0x%x y = 0x%x \n",
12             x, y);
13      return 0;
14  }
```

```
1   /* bar5.c */
2   double x;
3
4   void f()
5   {
6       x = -0.0;
7   }
```

**答：将 bar5.c 中的 double x 改为 static double x**
**或将 foo5.c 中的 int x 改为 static int x**

CSAPP 7.9 考虑下面的程序，它由两个目标模块组成：

```
1   /* foo6.c */
2   void p2(void);
3
4   int main()
5   {
6       p2();
7       return 0;
8   }
```

```
1   /* bar6.c */
2   #include <stdio.h>
3
4   char main;
5
6   void p2()
7   {
8       printf("0x%x\n", main);
9   }
```

当在 x86-64 Linux 系统中编译和执行这个程序时，即使函数 p2 不初始化变最 main，它也能打印字符串 "Ox48\n" 并正常终止。请解释原因。

**答：foo6.c 中的 main 作为强符号被定义，而 bar6.c 中的 main 定义为弱符号，会被解析到 foo6.c 中的 main 上，所以在 p2 中调用 main 与 bar6.c 中的变量 main 是否初始化无关**

CSAPP 7.10 a 和 b 表示当前路径中的目标模块或静态库，而 a→b 表示 a 依赖于 b，也就是说 a 引用了一个 b 定义的符号。对于下面的每个场景，给出使得静态链接器能够解析所有符号引用的最小的命令行（即含有最少数量的目标文件和库参数的命令）。

A. `p.o` → `libx.a` → `p.o`

B. `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a`

C. `p.o` → `libx.a` → `liby.a` → `libz.a` *and* `liby.a` → `libx.a` → `libz.a`

**答：**

**A: gcc p.0 libx.a**

**B: gcc p.o libx.a liby.a libx.a**

**C: gcc p.o liibx.a liby.a libx.a libz.a**

CSAPP 7.12 考虑目标文件 m.o 中对函数 swap 的调用（作业题 7.6）。

```
9:  e8 00 00 00 00          callq e<main+Oxe>    swap()
```

具有如下重定位条目：

r.offset = 0xa

r.symbol = swap

r.type = R_X86_64_PC32

r.addend = -4

A. 假设链接器将 m.o 中的 .text 重定位到地址 0x4004e0, 把 swap 重定位到地址 0x4004f8 。那么 callq 指令中对 swap 的重定位引用的值应该是什么？

B. 假设链接器将 m.o 中的 .text 重定位到地址 0x4004d0, 把 swap 重定位到地址 0x400500 。那么 callq 指令中对 swap 的重定位引用的值应该是什么？

**答：**

**A：当前指令地址：0x4004e0 + r.offset = 0x4004ea**

**下一条指令地址：0x4004ea + 5 = 0x4004ef**

**偏移量：0x4004f8 + (-4) - 0x4004ea = 0x5**

**B：同理可得重定位引用的值为 0x1d**

一位程序员编译运行下面两个文件：

```
----------------------------------------
main.c:

#include <stdio.h>

long a = 1;
const long b = 2;
long c;
long d = -1;

int main(int argc, char *argv[]) {
    printf("a: %p\nb: %p\nc: %p\nd: %p\n", &a, &b, &c, &d);
    printf("%ld\n", c);
    return 0;
}
----------------------------------------
data.c:

unsigned int c[2] = {...};
----------------------------------------
```

看到了下面的输出：

```
a: 0x601020
b: 0x400650
c: 0x601030
d: 0x601028
4294967297
```

注：省略了%p 输出的前导零

他原以为这些变量会一个一个按顺序分布，很明显他错了。请分析发生了什么。

a）可执行文件的符号表中，由 main.c 生成的符号有几个？

b） main.c 中定义的强符号有哪些，弱符号有哪些？

c）b 的地址为什么离其它变量那么远？

d） 为什么 c 的地址在 d 后面？

e）最后一行的输出 4294967297 是变量的值还是地址？为什么？

**答：**

**A. 5 个，a,b,c,d,main**

**B. 强：a,b,d,main 弱：c**

**C. 因为 b 是一个 const 常量，存储在在.rodata 段中，不在.data 段中**

**D. 因为 main.c 中的 c 为弱符号，被 data.c 中的 c 强符号定义，打印的时候输出的是强符号的地址，所以在 d 的后边**

**E. 地址，因为 c 是一个数组的指针，其指向一个地址空间的头部**